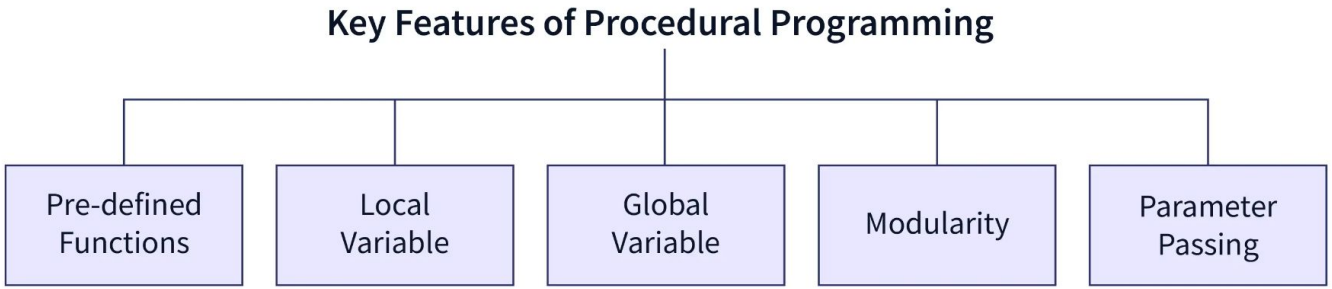
Object Oriented Programming

Motivation

IT560

Sourish Dasgupta
Assistant Professor, DA-IICT, Gandhinagar
https://daiict.ac.in/faculty/sourish-dasgupta

What happened so far? Procedural Programming



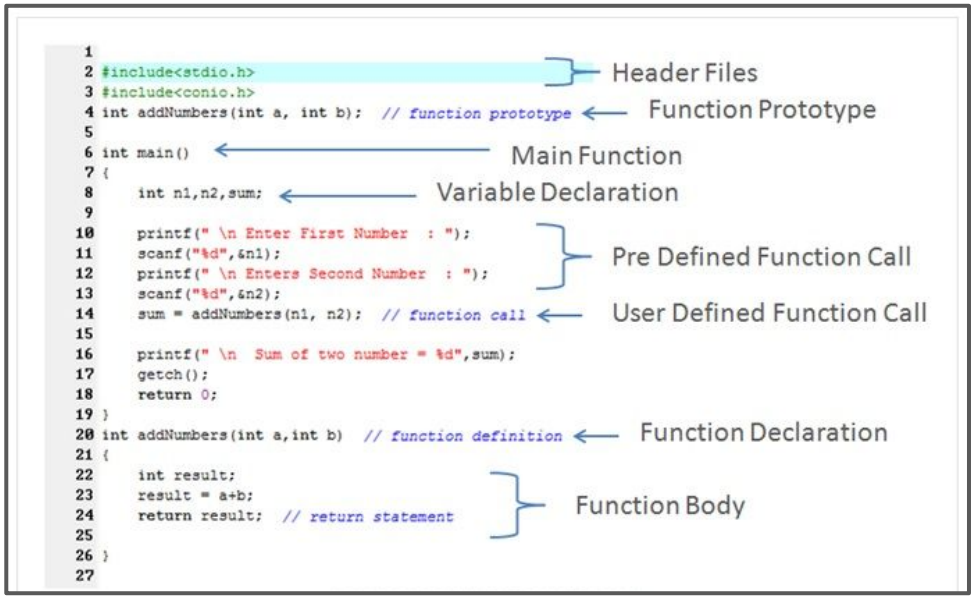
SCALER Topics

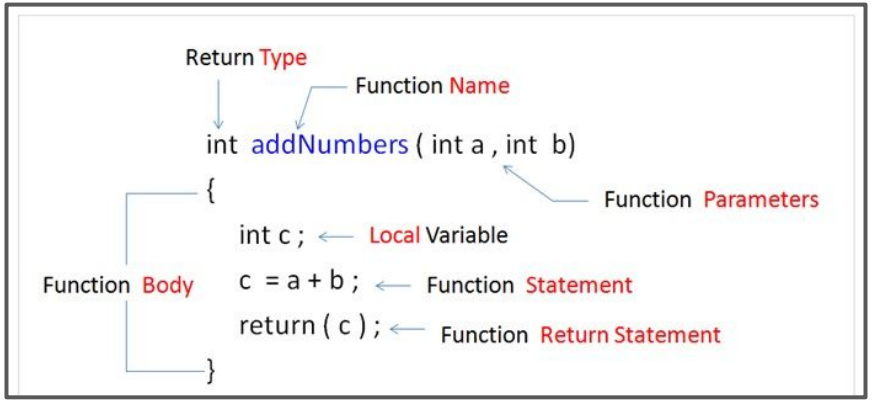
Procedural Programming: Structure

```
1
2  #include<stdio.h>                                   } Header Files
3  #include<conio.h>
4  int addNumbers(int a, int b);  // function prototype  <- Function Prototype
5
6  int main()                                          <- Main Function
7  {
8      int n1,n2,sum;                                  <- Variable Declaration
9
10     printf(" \n Enter First Number  : ");           } Pre Defined Function Call
11     scanf("%d",&n1);
12     printf(" \n Enters Second Number  : ");
13     scanf("%d",&n2);
14     sum = addNumbers(n1, n2);  // function call     <- User Defined Function Call
15
16     printf(" \n  Sum of two number = %d",sum);
17     getch();
18     return 0;
19 }
20 int addNumbers(int a,int b)  // function definition  <- Function Declaration
21 {
22     int result;                                     } Function Body
23     result = a+b;
24     return result;  // return statement
25
26 }
27
```

```
#include <stdio.h>                                   <- Function Parameters
int addNumbers(int a, int b);                        <- Function Prototype
int main()                                           <- Main Function
{
    ... .. ...
    sum = addNumbers(n1, n2);                        <- Function Call Statement
    ... .. ...
}
int addNumbers(int a, int b)                         <- Function Declaration
{
   ... .. ...
   ... .. ...
   return result                                     <- Function Arguments
}
```

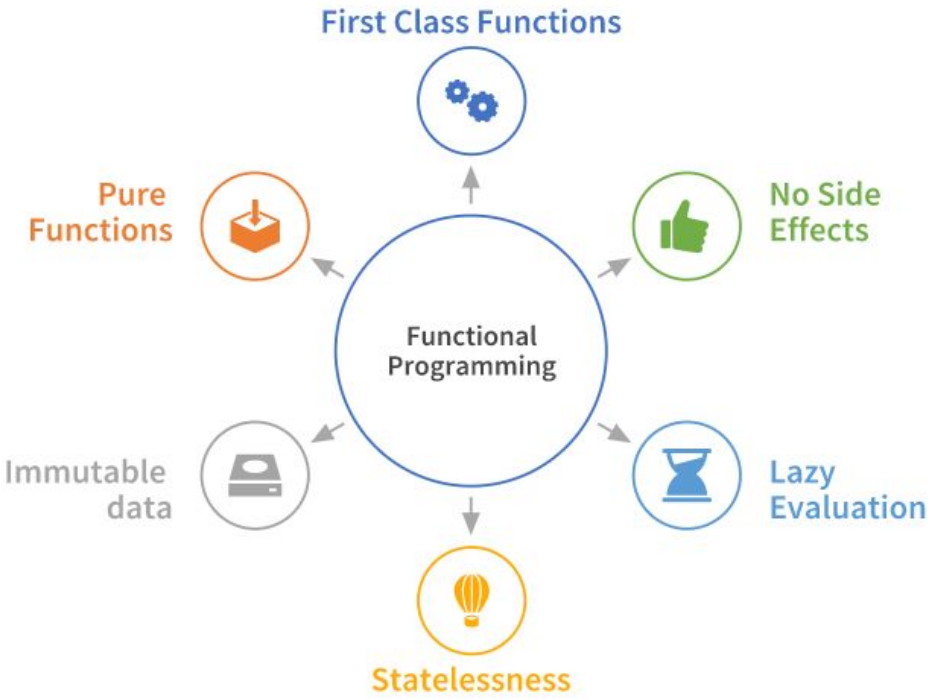
Procedural Programming: Structure





So then what's wrong with Procedural?

Design Principles of Functional Programming



Functional Programming: First-class Functions

```
#include <iostream>
#include <functional>

// A simple function that takes another function as a parameter

void executeFunction(const std::function<void(int)>& func, int value) {
    func(value); // Call the passed function with the given value
}

int main() {
    // Define a lambda function
    auto printSquare = [](int x) {
        std::cout << "The square of " << x << " is " << (x * x) << std::endl;
    };

    // Pass the lambda function to another function
    executeFunction(printSquare, 5);

    return 0;
}
```

Capture clause: no variable captured from the scope

First-class function

Achieving SoC via Pure & First-class Functions

```cpp
#include <iostream>
#include <string>

void processUserInput() {
    std::string userInput;
    std::cout << "Enter a number: ";
    std::cin >> userInput;

    // Validate the input
    for (char c : userInput) {
        if (!isdigit(c)) {
            std::cout << "Invalid input!" <<
std::endl;
            return;
        }
    }

    // Convert the input and calculate the
square
    int number = std::stoi(userInput);
    std::cout << "The square is: " << (number *
number) << std::endl;
}

int main() {
    processUserInput();
    return 0;
}
```

Procedural Style

```cpp
#include <iostream>
#include <string>
#include <optional>
#include <functional>

// Pure function to get user input std::string
getInput() {
std::cout << "Enter a number: ";
std::string input;
std::cin >> input;
return input;
}

// Pure function to validate input
std::optional<int> validateAndConvert(const
std::string& input) {
for (char c : input) {
if (!isdigit(c)) {
return std::nullopt;
  // Return an empty optional for invalid input
 }
}
return std::stoi(input);
// Return the converted number if valid
}

// Pure function to process
(e.g., calculate the square)
int calculateSquare(int number) {
return number * number;
}
```

```cpp
// Functional composition: Combine pure functions
void processInput(const std::function<void(const
std::string&)>& successHandler, const
std::function<void()>&errorHandler) {
auto input = getInput();
auto validatedInput =validateAndConvert(input);
if (validatedInput) {
successHandler(input);
// Call success handler with valid input
} else {
errorHandler();
// Call error handler
 }
}

int main() {
processInput([](const std::string& input){
int number = std::stoi(input);
int square = calculateSquare(number);
std::cout << "The square is: " << square << std::endl; },
[]() {
 std::cout << "Invalid input!" << std::endl; }
);
return 0;
}
```

Achieving Behavior Abstraction via Higher-order Functions

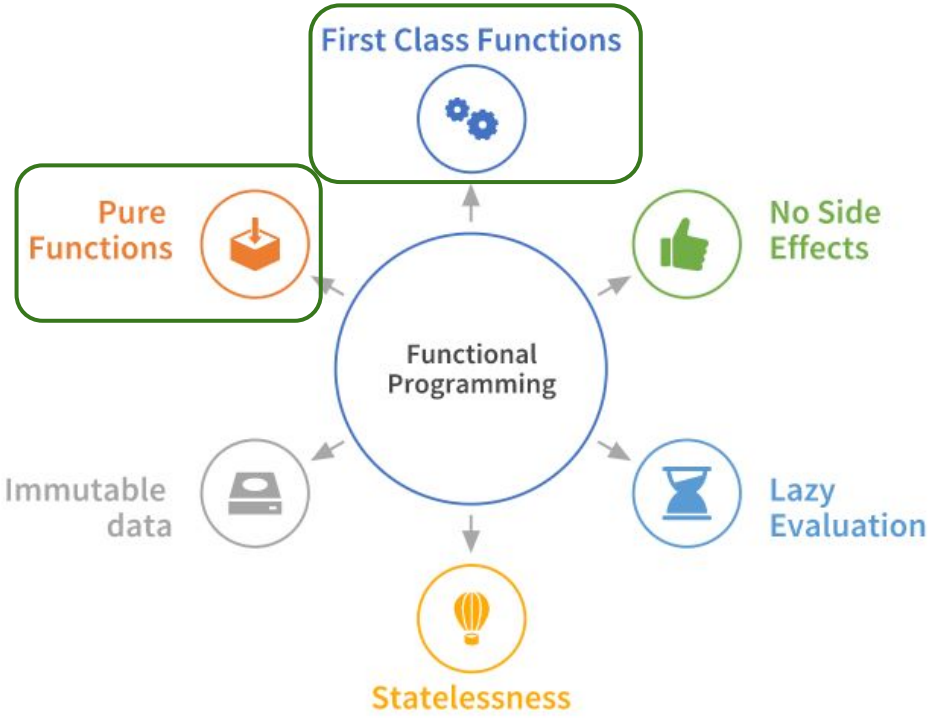
```
#include <iostream>
#include <vector>
#include <functional>

// Higher-order function for processing numbers
void processNumbers(const std::vector<int>& numbers,
                    const std::function<int(int)>&operation) {
    for (int num : numbers) {
        std::cout << operation(num) << " ";
    }
    std::cout << std::endl;
}

int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};

    // Pass different operations to processNumbers
    processNumbers(numbers, [](int x) { return x * x; });  // Print squares
    processNumbers(numbers, [](int x) { return x * x * x; });  // Print cubes
    processNumbers(numbers, [](int x) { return x + 10; });  // Add 10 to each

    return 0;
}
```

Achieving Dynamic Behavior via Functional Map

```
#include <iostream>
#include <functional>
#include <map>

// Functional map of operations
std::map<char, std::function<int(int, int)>> getOperations()
{
    return {
      {'+', [](int a, int b) { return a + b; }},
      {'-', [](int a, int b) { return a - b; }},
      {'*', [](int a, int b) { return a * b; }},
      {'/', [](int a, int b) { return b != 0 ? a / b : 0; }}
        // Handle division by zero
   };
}

void executeOperation(char operation, int a, int b,
const std::map<char, std::function<int(int, int)>>&
operations) {
    if (operations.find(operation) != operations.end()) {
        std::cout << "Result: " <<
        operations.at(operation)(a, b) << std::endl;
    }
```

```
else {
        std::cout << "Unknown operation!" <<
std::endl;
    }
}

int main() {
    auto operations = getOperations();

    char operation;
    int a, b;

    std::cout << "Enter operation (+, -, *, /): ";
    std::cin >> operation;
    std::cout << "Enter two numbers: ";
    std::cin >> a >> b;

    executeOperation(operation, a, b, operations);

    return 0;
}
```

Design Principles of Functional Programming



Functional Programming: Immutability vs. Statelessness

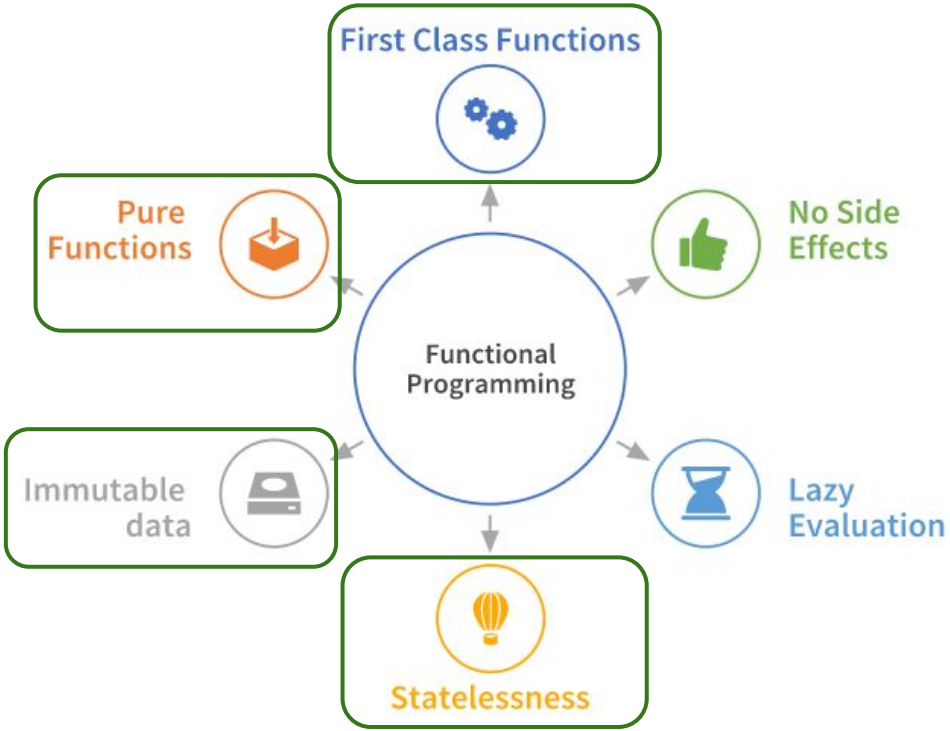
```
int globalCounter = 0;  // Mutable global state

int increment(int value) {
    return value + 1;
// Stateless function (no side effects, depends only on input)
}

// But the global state makes the system stateful:
globalCounter = increment(globalCounter);
```

```
const int immutableCounter = 5;

int incrementAndPrint() {
    std::cout << immutableCounter << std::endl;
// Has a side effect (stateful)
    return immutableCounter + 1;
}
```

Functional Programming: Combining both

```
#include <iostream>
#include <vector>
#include <algorithm>

// Stateless function operating on immutable data
std::vector<int> doubleNumbers(const std::vector<int>& numbers) {
    std::vector<int> result = numbers; // Copy input to preserve immutability
    std::transform(result.begin(), result.end(), result.begin()[,](int x) { return x * 2; });
    return result;  // Return a new vector without modifying the original
}

int main() {
    const std::vector<int> numbers = {1, 2, 3, 4, 5}; // Immutable input
    std::vector<int> doubled = doubleNumbers(numbers);// Stateless function
    for (int num : doubled) {
        std::cout << num << " ";
    }
    return 0;
}
```

Design Principles of Functional Programming

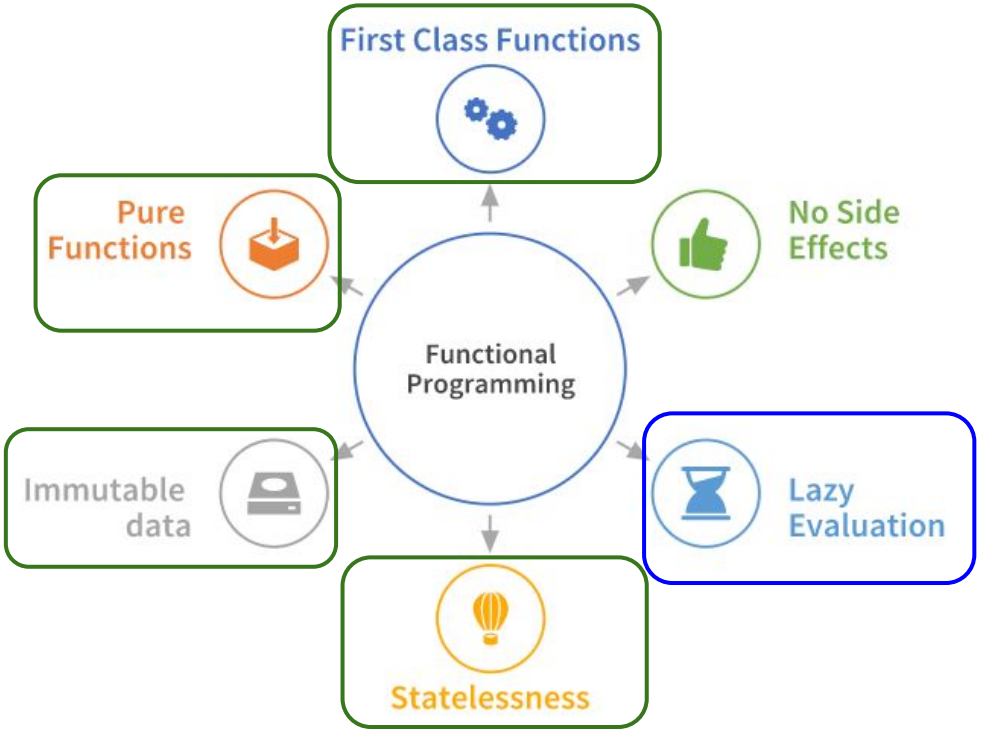


Lazy Evaluation via Generators

```cpp
#include <iostream>
#include <functional>

#State-less Sequence Generator
class LazySequence {
public:
    LazySequence(int start, int step) :current(start), step(step) {
}
// member variables directly initialized before the body of the constructor is executed

    int next() {
        int value = current; // Capture the current value
        current += step; // Increment for the next value
        return value;
        // Generates the next value only when needed
    }

private:
    int current;
    // Tracks the current value in the sequence
    int step; // The step size for the sequence
};
```

```cpp
int main() {
    LazySequence sequence(0, 2);
    // Start at 0, increment by 2

    // Generate and print the first 5 elements lazily
    for (int i = 0; i < 5; ++i) {
        std::cout << sequence.next() << " ";
        // Outputs: 0 2 4 6 8
    }

    return 0;
}
```

Design Principles of Functional Programming



The final bit - Side-effects

```
int addAndPrint(int a, int b) {
    int result = a + b;
    std::cout << "The result is: " << result << std::endl;  // Side effect: I/O
    return result;
}
```

```
#include <fstream>

void writeFile(const std::string& filename, const std::string& content) {
    std::ofstream file(filename);
    file << content;  // Side-effect: Writes data to a file (external system interaction)
}

int main() {
    writeFile("example.txt", "Hello, file!");
    return 0;
}
```

Removing Side-effects

```
void printResult(int result) {
    std::cout << "Result: " << result << std::endl; // Isolated side-effect
}

int add(int a, int b) {
    return a + b; // Pure function
}

int main() {
    int result = add(3, 4);
    printResult(result); // Side effect is isolated here
    return 0;
}
```

Functional Programming: Key takeaways

1. **Dynamic Behavior**: Functions can be dynamically *assigned or modified at runtime*

2. **Separation of Concerns (SoC)**: *Decouple logic* (what to do) from implementation details (how to do it)

3. **Encapsulation of Behavior**: Functions *encapsulate behavior*, allowing them to be treated as reusable, interchangeable logic units.

4. **Runtime Composition**: *Design complex behaviors or pipelines* (without duplicating code) by composing or chaining functions dynamically.

Then why OOP?

We will see next ….